2009-06

Application of VMware VProbes to debugging of a segmentation based separation kernel

Sanders, Kyle

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

APPLICATION OF VMWARE VPROBES TO DEBUGGING OF A SEGMENTATION BASED SEPARATION KERNEL

by

Kyle Sanders

June 2009

Thesis Advisor:	Cynthia E. Irvine
Second Reader:	David J. Shifflett

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
16–6–2009	Master's Thesis	2008-12-01—2009-6-19

4. TITLE AND SUBTITLE

Application of Vmware Vprobes to Debugging of a Segmentation Based Separation Kernel

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Kyle Sanders

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Navy

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Debugging is a useful technique in all aspects of software development, including that of operating systems. Because they provide low level interfaces to the hardware, operating systems are particularly difficult to debug. There is little room to add abstraction between the computer hardware and the executing operating system software. Many debuggers are intimately tied to the system's memory model, compiler, and loader. For specialized operating systems, a good debugger may require a focused development process. However, virtualization allows new tools to be introduced to support operating system debugging. This research explores the potential to use VMware VProbes to debug a Least Privilege Separation Kernel (LPSK) that is hosted as a guest operating system by the VMware virtual machine monitor. Several general tools were developed to help developers use VMware VProbes. In particular, a simple way to use the symbol table that makes references to memory easier to manage. As a result, the state of the target operating system can be inspected upon access to one or more memory addresses. The tools and techniques were tested on the LPSK; however, they may be applied to a wide range of operating systems hosted by VMware.

15. SUBJECT TERMS

TCX, Least Privilege Separation Kernel, Debugging, VMware

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	84	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

APPLICATION OF VMWARE VPROBES TO DEBUGGING OF A SEGMENTATION BASED SEPARATION KERNEL

Kyle Sanders
Civilian, Naval Postgraduate School
B.S. Western Oregon University, 2007

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2009

Author: Kyle Sanders

Approved by: Cynthia E. Irvine
Thesis Advisor

David J. Shifflett
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Debugging is a useful technique in all aspects of software development, including that of operating systems. Because they provide low level interfaces to the hardware, operating systems are particularly difficult to debug. There is little room to add abstraction between the computer hardware and the executing operating system software. Many debuggers are intimately tied to the system's memory model, compiler, and loader. For specialized operating systems, a good debugger may require a focused development process. However, virtualization allows new tools to be introduced to support operating system debugging. This research explores the potential to use VMware VProbes to debug a Least Privilege Separation Kernel (LPSK) that is hosted as a guest operating system by the VMware virtual machine monitor. Several general tools were developed to help developers use VMware VProbes. In particular, a simple way to use the symbol table that makes references to memory easier to manage. As a result, the state of the target operating system can be inspected upon access to one or more memory addresses. The tools and techniques were tested on the LPSK; however, they may be applied to a wide range of operating systems hosted by VMware.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

This material is based upon work supported by the National Science Foundation, under grant No. DUE-0414102. In addition, partial support for this work was provided by the Office of Naval Research, the Nation Reconnaissance Office and the National Science Foundation, under grant CNS-0430566. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation, The Office of Naval Research or the National Reconnaissance office.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 Motivation

Developing and maintaining operating systems has become a task that requires advanced skills and knowledge. Developers must thoroughly test and debug an ever expanding code base that is adapted to more and different hardware. This creates the problem of systemic debugging. Systemic debugging views the system as a whole instead of debugging a single program. Technologies such as the Quick Emulator (QEMU) and the GNU Debugger (GDB) are currently used for debugging of operating systems. However, these tools have limitations inherent to their design.

Using software debuggers such as GDB offers flexibility but comes at a cost of robustness. Flexibility is achieved by the debugger allowing users to stop execution flow, step through code and change memory values. Robustness is lost because software debuggers change the state of the system which is being debugged. They are generally configured into the kernel as a device driver and are difficult to interface with.

Recent advances in virtualization allow operating systems to be run on top of other operating systems. Specifically, the base operating system runs a virtual machine monitor. Since the virtual machine monitor is at a lower abstraction layer, and manages the virtual machines, it can be inspected without instrumentation.

Using virtualization to support debugging offers the benefits of existing techniques while reducing many of their drawbacks. The virtualized environment acts as a layer of abstraction between the hardware and the operating system. Information can be gathered without affecting the state of the operating system in the virtual machine. In addition, the technology does not require the acquisition of new hardware or modification of the target operating system. Unlike GDB the increase in functionality does not reduce the level robustness of the target.

VMware Workstation 6.5 provides a tool, called VProbes, for operating system debugging using virtualization. As previously discussed, VProbes has many advantages relative to many existing debugging technologies and addresses a number of their shortcommings. This research explored VProbes and its application as a debugger for a Least Privilege Separation Kernel.

1.2 Purpose of Study

The objective of this thesis is to determine whether VMware VProbes can be used as a debugging mechanism for the Naval Postgraduate School (NPS) Least Privilege Separation Kernel (LPSK). Additionally, this thesis will develop a general purpose guide to using VMware VProbes with boutique operating systems. This thesis examines the following questions:

1. How does VMware Vprobes compare to existing debugging technologies?
2. Can this debugging tool be applied to a segmentation-based separation kernel?

This thesis presents an organized framework for answering these questions. Background material on virtual machine monitors and existing debugging techniques will be gathered. This will provide a basis for comparison between VMware VProbes and existing debugging technologies. Second, the LPSK will be instrumented with VMware VProbes. Furthermore, several tools will be developed to assist LPSK developers when using VMware VProbes as a debugging mechanism. The applicability of these techniques and tools to other boutique operating systems will be discussed.

1.3 Organization of Thesis

The organization of this thesis is as follows:

Chapter 1 illustrates the motivation and purpose for this thesis. Furthermore, the problem of debugging operating systems is presented.

Chapter 2 provides background information on different components of this thesis. This includes the following items: virtualization, a debugger taxonomy, existing debugging techniques, VMware VProbes, the LPSK, and the Open Watcom build tools.

Chapter 3 presents concepts for using VMware Vprobes with the LPSK. Furthermore, a framework for generating symbol tables is presented. Finally, validation tests for ensuring the correctness of VMware VProbes are presented.

Chapter 4 covers related work on operating system debugging. Technologies such as: DTrace, K42 Kernel, ICE, and GDB will be presented.

Chapter 5 concludes this thesis with a summary and future work possibilities.

CHAPTER 2:
Background

This chapter presents virtual machines, a debugger taxonomy, comparison of debugging technologies, the Trusted Computing Exemplar (TCX) project, Open Watcom build tools, and symbol tables. These topics provide a background for this thesis.

2.1 Virtual Machines

Any discussion of virtual machines requires precise definitions of key terms. For these, we will use Goldberg's definitions [8]. First is the concept of an `environment`. The environment is simply the target system to be duplicated [8]. As Goldberg states, operating systems and programs that run on the bare hardware must run identically on the virtual machine with the same effect [8]. A key notion behind the concept of virtual machines is that most instructions must execute directly on the target's central processing unit [8]. Goldberg calls this concept the `implementation`. Executing instructions directly provide two benefits: performance will improve and uniformity between the host and the virtualized environment will be reinforced.

The concepts of environment and implementation help define a virtual machine [8, 23]. Goldberg's definition of a virtual machine is

> a hardware software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute directly in the host processor in native mode [8].

The environment is the host processor being duplicated while the implementation is the means by which the statistically large number of instructions are executed on the underlying hardware.

Goldberg's working definition states that a virtual machine is a software-hardware combination. The software component is known as the Virtual Machine Monitor (VMM) [8]. According to Goldberg, Virtual Machine Monitors are of two kinds: self virtualizing and family virtualizing [8]. Self virtualizing systems are Virtual Machine Monitors that present an exact copy of the underlying hardware architecture within its virtual machine. Alternatively, family virtualization means that the Virtual Machine Monitor presents the same system family as the underlying system. An example of family virtualization would be a Virtual Machine Monitor that virtualizes

		Virtualized Environment	
Underlying System	Hardware Extended	Hardware Type I VMM Type II VMM	Extended Type I EMM Type II EMM

Table 2.1: Table illustrating different Virtual Machine Monitor types. Taken from [8]

a Pentium 3 processor while the underlying host is a Pentium 4 processor. Both of these CPUs are Intel processors and, more specifically, support the x86 instruction set.

2.1.1 Types of Virtual Machines

This section presents the different types of virtual machines. Goldberg presents two different types of virtual machines: Type I and Type II [8]. Furthermore, the difference between an Extended Machine Monitor and a Virtual Machine Monitor is established.

Type I Virtual Machine Monitors execute on the bare hardware of the underlying system [8]. This means that the VMM must provide facilities for managing the hardware resources. This type of Virtual Machine Monitor is likely to be much larger than Type II virtual machines because of the resource management support it must provide [8]. Type II Virtual Machine Monitors run under the control of an existing operating system and provide a virtual machine [8]. This type of VMM has a much smaller code base because the supporting operating system provides extensive resource and hardware management.

When Type I and II Virtual Machine Monitors are compared, there are advantages and disadvantages to each. Type II VMMs are smaller and allow users to have a preferred operating system as the platform [8]. However, if there is a need for a large number of virtual machines with no preference for a platform operating system, Type I Virtual Machines are better to meet these requirements. Specifically, Type I Virtual Machine Monitors will be in sole control of system resources and can allocate them in a fashion that is more conducive to large numbers of virtual machines.

Goldberg also presents the concept of an Extended Machine Monitor (EMM). Compared to a Virtual Machine Monitor an Extended Machine Monitor provides a more abstract view of the system. The underlying target architecture is changed to make it more convenient to use [8]. An example of this type of system would be the Linux kernel. This operating system provides programmers with an abstract view of the underlying architecture. In fact Type II

Virtual Machines run on Extended Machine Monitors, namely the supporting operating system [8]. This statement also implies that existing operating systems are Type I extended machines [8]. Type II Extended Machine Monitors run on an existing operating system to provide the interface of an existing operating system. Table 2.1 illustrates these differences.

2.1.2 Virtual Machines and Other Technologies

Technologies exist that provide some of the same functionality as Virtual Machine Monitors but differ greatly in their implementation. This section highlights some of these technologies and illustrates how they differ from Virtual Machine Monitors. Many of these technologies are used in this thesis as a comparison to VMware. The distinctions made will be based upon Goldberg's classifications.

`Emulation` provides an interface for guest operating systems similar to that presented in virtual machines. However, emulation is implemented very differently. Emulation provides an abstract view of the underlying hardware that does not need to correspond to what is actually present [8]. This means that code written for a SPARC processor could be run on an Intel Pentium 4 processor. Since we can recall from Section 2.1 that Virtual Machines must be of the same processor family (or identical), emulation provides a completely new functionality. However, this technology does not allow instructions to be executed on the bare hardware and consequently suffers from performance issues. In terms of our keywords, emulation can provide the environment that virtual machines do, but cannot provide the implementation.

Since emulation does not reflect the underlying hardware, debugging low level problems can be problematic. Using our previous example, debugging becomes problematic when code written for a different architecture is run on an emulator. There is a disconnected layer of indirection before execution. A bug in the emulation software would manifest itself as a bug in the processor. The reverse situation is that a bug in the code may only happen if the emulation acts in accordance with the processor.

2.2 Debugging

`Debugging` is defined as the act of methodically searching for and reducing bugs in a target program. The term `debug` is often credited to Rear Admiral Grace Hopper who found a moth in a computer that was not functioning properly. It is important to note that the aforementioned definition does not mention any one technique or technology. Debugging can even be as simple

as using such fundamental techniques as source code review. In fact, the most widely used debugging technique is to insert print statements into code.

In certain situations it is desirable to have a better mechanism for debugging. Developers have recognized this need and have created tools known as debuggers. Debuggers show information about the currently executing process. This information includes values of variables, memory values throughout the processes memory, one or more lists of active functions and processor register values [14]. Debuggers run in their own process space to protect them from errors in the target program. The use of debuggers gives developers a general purpose mechanism to explore running processes, and help developers understand why the target process is producing incorrect results [14].

Traditionally debuggers take two distinct forms; source level debugger and machine level debugger. Source level debuggers debug a target program in terms of the source code developed. As the debugger progresses through the code's instructions, it will show the progress in the source code. Machine level debuggers view the target programs in their compiled form. While the machine code can be mapped to the source code, the developer must do this by hand. As the debugger progresses, progress will be shown in the compiled code.

2.2.1 Debugger Taxonomy

Dbx originally was a debugger for the Pascal language. In "The Evolution of Dbx", Linton outlines several parts of the Dbx debugger [14]. There is no standard way to construct a debugger, but the division presented by Linton is a good general model for debuggers. This section will expand Linton's division into key functionalities and aspects of debuggers. This set of debugger features will provide a basis to compare existing technologies to VMware VProbes. Debuggers incorporate the following functionalities: view and change memory, control program flow, display information in symbolic form, and an interface to control the debugger.

The ability to view and change memory is a key functionality for debuggers. Viewing memory allows developers to extract information from the running process without embedding print statements. This allows developers to avoid adding large amounts of debugging code to target systems and prevents the act of printing information from disturbing the target's process state. Changing values in memory enables developers to perform "what if" scenarios. For example, what if this variable is set to a given value? Will it result in a buffer overflow? These questions can be answered without developing code to test each scenario. The debugger enables the

programmer to insert values into selected memory locations and see the resulting effects of the change.

Controlling program flow provides developers with a powerful mechanism. Execution control allows developers to stop programs at any point and examine the state of the process [7]. `Breakpoints` allow developers to suspend execution when a certain point in the program is reached [14]. For example, a developer may select the address of a function suspected of causing problems. When the program reaches that address it will stop executing allowing for inspection of the program. `Stepping` allows the debugger to execute a certain number of instructions then stop execution [14, 7]. Depending on debugger functionality, developers may specify how many instructions to step by. The GNU Debugger allows developers to step by function calls.

Displaying information in symbolic form provides a key user interface to the developer. Instead of viewing opcodes of the compiled binary, a programmer can see the mapping to the source code [14]. This type of functionality is only present in source level debuggers. The distinction between source level debuggers and machine level debuggers is described in Section 2.2.

The control interface for debuggers is a user interface aspect to the tool. GDB and Dbx both use command line programs to provide a control mechanism [14, 7]. While this is one option for providing a control mechanism, many source level debuggers are built directly into an Integrated Development Environment. This approach provides more visualization for the developer. The key aspect of either approach is that it provides a control interface for developers to use the aforementioned functionalities in a convenient manner.

2.3 Debugging Technologies

Currently there are many different techniques for debugging programs. As mentioned in Section 2.2, this includes embedding print statements into code. This section will focus on presenting technologies that are used to debug an entire system, often referred to as systemic debugging [4]. Furthermore the taxonomy from Section 2.2.1 will be used as a framework for describing each technology.

2.3.1 QEMU

The Quick Emulator or QEMU is an open source project that provides emulation for main stream operating systems [20]. It provides two modes of operation: full system emulation and

	QEMU	GDB	VProbes
Viewing Memory	X	X	X
Changing Memory	X	X	
Breakpoints	X	X	
Stepping	X	X	
Symbolic Form	X	X	X
Control Interface	X	X	X

Table 2.2: Table illustrating different debugging technologies and functionalities present

user mode emulation. Full system emulation allows a user to run an entire operating system in user space similar to VMware. User mode emulation allows programs compiled for different architectures to be run, regardless of target architecture. This section will focus on the full system emulation as it is suited for systemic debugging. It is important to note the difference between emulation and virtual machines, see Section 2.1.2 for a more in depth discussion.

QEMU by itself does not constitute as a debugger. It provides the emulation mechanism in which code for many different architectures can be run on a single machine [20]. Since the entire machine runs within the confines of the QEMU process, existing debuggers such as GDB can be used [20]. Thus, all functionalities present in GDB are present in QEMU. Table 2.2 outlines these functionalities.

2.3.2 GDB

The GNU Debugger or GDB is a popular open source debugger [7] and is one of the predominant debuggers for Unix type systems. GDB can be split into three main components: the user interface, symbol handling, and target system handling. This software has four primary functions that promote debugging: start the target program with anything that can affect behavior, set breakpoints to force the program to stop given a certain condition, view state of the target program in a stopped state, and change values in the target program [24]. One can see how these four main functions fit into the debugger taxonomy discussed in Section 2.2.1.

Table 2.2 shows its many key functionalities. In addition to supporting application-level debugging, Linux developers also use GDB inside of the kernel to debug troublesome code. This requires the debugger to run inside of the kernel itself, which creates the possibility that the debugger is changing the state of the kernel. Execution of the debugger in the kernel also suffers from being difficult to set up and configure. Running GDB inside of the kernel requires the use

of two machines. One machine will act as the target for debugging. This machine will have the GDB server compiled as part of the kernel. The second machine acts as a client, communicating to the GDB server executing on the target system.

2.3.3 VProbe Debugging

VProbes provide a window through the VMware Virtual Machine Monitor to view the target operating system [11]. From Section 2.1.1 one can note that VMware is a Type II virtual machine. As such, many different types of operating systems can run on top of the virtual machine so long as the code is meant to run on an Intel processor. This provides boutique operating system developers with the ability to debug code flexibly.

VMware lists the following design goals for VProbes; safe, dynamic, static, operating system independent, and free with respect to system resources when disabled [11]. The safety design goal allows VProbes to be instrumented without affecting the virtual machine state of the target [11]. The dynamic aspect of VProbes allows them to be inserted or removed in the target machine during runtime [11]. The static aspect of VProbes means that predefined probes can be invoked when certain events occur during the target systems execution [11]. Operating system independence allows for different operating systems to be inspected with VProbes [11]. VProbes incurs no cost in CPU cycles or memory when disabled. This allows production systems to not be hindered by its use.

VProbes are implemented using a custom scripting language called VP script [11]. The use of a scripting language enables the VProbes to be used in a general purpose manner. Developers can write functions which build upon one another as for any modern programming language. No constructs from the underlying OS are required for their use [11]. This ensures the VMware VProbes design goal of operating system independence.

Given the debugger taxonomy in Section 2.2.1, VProbes does not have many of the functionalities of current debuggers, see Table 2.2. It only supports viewing of memory, use of symbolic formats and a control interface. This is an artifact of the VProbes design goals. For the VProbes to be safe and not be noticeable by the target system they cannot control program flow. Furthermore, VProbes cannot change memory because of the aforementioned design goal. However, the VProbes can view systems from the Virtual Machine Monitor without interfering with their state, unlike other debugging technologies that must either run on the same operating system or does not allow instructions to be executed on the underlying hardware.

2.4 Symbol Tables

`Symbol tables` provide mappings between a symbol name and symbol address. A symbol is a name for an entity in the memory of the computer. For example, variable and function names are considered symbols. Using symbol tables allows developers to use mnemonic devices for memory locations instead of numerical addresses.

VProbes uses linear memory addresses as probe points. To make the VP Scripts easier to write VMware allows users to specify a symbol table for the target operating system. The addresses in the symbol table must be linear addresses for the symbols. To assist in debugging the Least Privilege Separation Kernel, a symbol table will be built with mappings from each symbol to the linear address the symbol resides in.

2.5 Operational VProbes

There are two types of VProbes [11]: static and dynamic. Static probes are predetermined points of virtual machine execution [11]. Static probes include predetermined timing intervals and interupt requests. Dynamic VProbes are user-specified probe points [11]. All dynamic VProbes are installed to trigger upon references to linear addresses of the target operating system.

There are three types of dynamic Vprobes; read, write, and execute [11]. Read VProbes watch a linear address in memory for read operations [11]. Once the VMM detects a read from the linear address the read VProbe will be executed. Write VProbes operate in the same fashion as read VProbes, except the VProbe is executed when a write to the linear address is performed [11]. Execution VProbes monitor a linear address and execute the VProbe when that address is fetched for execution [11].

2.6 Trusted Computing Exemplar

The Trusted Computing Exemplar (TCX) project [12] was created in response to the atrophying abilities of programmers to build high assurance systems. The project has four main goals [12]: creation of a prototype framework for rapid high assurance development; reference-implementation of trusted computing; evaluation of the component for high assurance; and open dissemination of all artifacts of the project.

At the heart of the TCX project is the Least Privilege Separation Kernel (LPSK) [12]. The primary focus of the kernel is to provide data-domain and process separation [12]. Furthermore,

primitive kernel level functionalities are also provided by the kernel [16]. The LPSK supports a fixed number of resources and subjects, both of which are statically allocated to partitions.

An instance of the LPSK is configured in a static fashion via the `Configuration Vector` [12]. The configuration vector is read during LPSK initialization. The processes to be executed are specified in the configuration vector and the kernel will set up the partitions accordingly. Furthermore, process/resource bindings are specified in the configuration vector, which spell out the interaction processes may have with other processes and resources [12]. These bindings are static and will be applied until the system has been shutdown.

Processes are mapped to partitions in the LPSK. One way for two processes to communicate is through shared memory segments, which are defined in the configuration vector. All system-internal communication channels between processes are known before system execution begins.

Unlike most modern operating systems, the LPSK uses all four Intel hardware privilege levels (PL). This permits enforcement of a privilege policy. The only way for execution in a chosen privilege level to invoke a lower level is through controlled interface such as a call gate.

The LPSK uses the Large memory model [12]. This allows a process to have multiple memory segments in the address space of a subject. Thus, a subject is an active entity in a particular privilege level. There can be multiple code and data segments in a privilege level. In order for the LPSK executable to define multiple code and data segments, the Linear Executable (LX) [10] format is used.

As of June, 2009 the LPSK provides default modules for all hardware privilege levels. Privilege level 3 is the application layer, and the LPSK developers provide several test applications [21]. This thesis uses the TSM application for testing purposes. The SecureCore Operating System (SCOS) and SecureCore Security Services (SCSS) modules run in PL2 and PL1, respectively [22, 5]. The LPSK kernel executes in PL0 [22].

2.7 Open Watcom

The TCX project uses the Open Watcom compiler [18] and its associated build tools [19]. This set of build tools supports the LX executable format used for the LPSK.

The linker turns abstract names into concrete locations [13]. For example, it turns the names of functions into the address of that function [13]. The Open Watcom linker [19] provides this functionality for the TCX project.

The Open Watcom linker has the ability to output the work it performs by creating a MAP file. The MAP file contains a great deal of information about the binary that was compiled. More details on the MAP file will be presented in Chapter 3.

2.8 Summary

This chapter has presented Virtual Machines, a debugger taxonomy, comparison of current debugging technologies, VMware VProbes, symbol tables and the TCX project. The next chapter will present how to implement and use VMware VProbes for the LPSK. Furthermore, the next chapter will verify the output received from VProbes while debugging the LPSK.

CHAPTER 3:
Application of VMware VProbes to The LPSK

This chapter outlines the methodology used to implement VMware VProbes as a debugging mechanism for the NPS Least Privilege Separation Kernel (LPSK). The organization of this chapter will progress in a linear fashion. We start with a brief overview of VP script and MAP files. This is followed by an in depth look into the construction of symbol tables for the LPSK. Finally, two verification tests to ensure that data extracted using VProbes is correct when used on the LPSK are described.

3.1 VP Script

VP Script is the instrumentation mechanism in which a probe can be loaded to trigger on a chosen linear address [11]. These probes can execute when said linear address is executed, read or written to. The scripting language has a syntax similar to that of lisp [11]. The language provides the ability for users to define their own variables and functions. Furthermore, several built-in functions are provided as well as global variables.

The language provides built in global variables that represent the values of CPU registers at the time the VProbe executes. Included in the many natively supported functions is the ability to read the contents of a specific linear memory address. However, the language does not allow for looping and recursion can only go 25 call levels deep. This is to ensure that the VProbe finishes executing.

A full description of VP Script is outside of the scope of this thesis. The VMware VProbes Programming reference provides the entire definition for the VPscript language [11].

3.2 MAP Files

During the build process, the Open Watcom linker can generate MAP files [19]. This is enabled by specifying the MAP option or ”-M” shorthand [19]. The current LPSK build process has the MAP enabled for default builds. Thus, for every binary compiled using the LPSK makefiles a MAP file is generated. An excerpt from an example MAP file can be seen in Figure 3.1.

A given MAP file contains a wealth of information about the binary for which it was generated. MAP files include information about the following aspects of the binary: Groups, Segments,

```
...
Address         Symbol
=======         ======
Module: kernel_ini1.o(/boot/tcx_kernel/kernel_ini1.asm)
0001:00000028  __halt
0001:0000002b  _asm_init_video
0001:00000057  __asm_put_char
0001:00000093* _get_ax
...
Module: kmem.o(/boot/tcx_kernel/kmem.c)
0001:0000b083+ _get_aligned_address
0001:0000b449  _kmem_init
0001:0000b4bc  _kmem_preallocate
0001:0000b6fa  _kmem_allocate
0001:0000b9d1+ _kmem_free
...
```

Figure 3.1: Example excerpt from a map file of the LPSK binary.

Memory Map, Module Segments, Imported Symbols and Linker Statistics. Each map file is for one module, where a module represents a file that is loaded into memory at boot time. For the purposes of building symbol tables for the LPSK only the memory map will be covered in this section.

The memory map provides a mapping between symbol names and their locations within a chosen segment. The symbol names include function names and variable names. For example, Figure 3.1 shows the `_kmem_allocate` symbol at address `0001:0000b6fa`. The `0001` portion of the address indicates the segment in which the symbol resides. The `0000b6fa` portion of the address is an offset into that segment.

3.3 TCX Symbol Tables

The kernel symbols will be loaded into locations specified by the kernel binary. From Figure 3.1 we know that functions reside in segment 1. To retrieve the starting address of segments as specified in the binary, the Open Watcom `wdump` tool can be leveraged. Once the linear address for the beginning of segment 1 has been extracted with `wdump`, it can be added to the offset specified in the MAP file. Extracting the linear base addresses for segments using `wdump` is covered in Appendix A.2.

To build a symbol table for the entire system it will require symbols from code in all four privilege levels. This requires linear addresses for the base location of each segment in memory.

Once this base address has been harvested, it can be added to the offset of each symbol in the Memory map. The resultant addresses will be the location of that item in memory as a linear address.

It is important to note that symbol tables are not required for VProbes to work with the LPSK. However, without the use of a mnemonic device such as a symbol table it becomes difficult to deal with all the different locations of functions and variables. As such, much attention will be paid to generating symbol tables for the LPSK and its applications.

3.3.1 LPSK Loader

As of April 2009, the TCX project had no support for reading files from disk. Thus the LPSK relies on the Grand Unified Bootloader (GRUB) [15] to read all of the files from disk. Once read into memory, the bootloader hands execution to the loaded LPSK. The LPSK is loaded into the linear address space specified in the `lpsk_kernel` executable. Furthermore, GRUB is a Multiboot [17] compliant bootloader which specifies a `module_t` data structure with information about each file read from disk.

The remaining binaries are loaded into memory using the kernel's loader. The kernel loader operates by first locating the binary to load into memory. Once located, the loader will allocate memory to copy the binary into. However, for each binary there are multiple allocation operations as the code, data and stack each get their own segments in memory.

To locate each binary in memory the kernel uses the `find_module` function. Figure 3.2 shows the prototype for the `find_module` function. This function will search through the Multiboot `module_t` data structure for the module specified by the parameters.

Once the starting address for the binary file has been located, it will be read and loaded into the appropriate privilege level according to the configuration vector. Each object in the binary will be loaded into separate regions in memory. To allocate this memory, the function `kmem_allocate` is called. Each individual binary, in general, will allocate three different segments of memory. These memory segments are for the code, data, and stack. Each time the `kmem_allocate` function is called, the starting linear address of the segment will be an output parameter.

The loader performs many more actions than those presented in this brief section. However, the portion necessary for creating symbol tables has been presented. Furthermore, the functions

```
int find_module(module_t *module_data, int  num_modules, char *path,
                int len, unsigned int *start_addr, unsigned int *size)

int kmem_allocate(unsigned int mem_size, unsigned int alignment,
                  unsigned int *start_addr)
```

Figure 3.2: Function prototypes for the find_module and kmem_allocate functions.

presented in this section have many more parameters than has been presented. The remainder of the loader and its functions is outside the scope of this thesis.

3.3.2 Harvesting Offset Addresses

As mentioned in Section 3.3 the addresses in the MAP file are relative to the segment they are present in. For a complete linear address to the symbol, the base of the memory segment must be obtained. During the loading process, as discussed in Section 3.3.1, a linear address is allocated for each segment loaded.

There are several ways in which linear addresses can be harvested. The kernel source code can be modified to display the start address every time the `kmem_allocate` function is called. Alternatively, a VProbe script can be configured to gather the starting address each time the `kmem_allocate` function is called. For the purposes of this thesis the second method will be used. It has the advantage of not requiring the source code of the kernel to be modified.

For the VProbe to execute properly, it must have the following information before the Virtual Machine is turned on: the linear address of the kernel stack and the linear address of `kmem_allocate`. The stack address is needed because parameters are passed on the stack in right to left order. The starting address of `kmem_allocate` is used to instrument the probe as an execution VProbe.

As previously mentioned in Section 3.1 all CPU registers are exported to VProbes as global variables. When retrieving parameters off the stack the ESP register marks the end. Each parameter can then be accessed by subtracting the variable position from the ESP address. However, the ESP register will only be relative to the current stack segment and not provide a linear address that can be accessed from a VMware VProbe. This is why the linear address of the stack segment needs to be provided to the VProbe. Once this is known, parameters can be accessed by adding the linear stack address to the ESP offset.

```
#wdump lpsk_kernel
...
                                  Object Table
==========================================================================
object 1: virtual memory size              = 0000C21FH
          relocation base address          = 00100000H
          object flag bits                 = 00002005H
          object page table index          = 00000001H
          # of object page table entries   = 0000000DH
          reserved                         = 00000000H
          flags = READABLE|EXECUTABLE|BIG

object 2: virtual memory size              = 0000E8D4H
          relocation base address          = 00110000H
          object flag bits                 = 00002003H
          object page table index          = 0000000EH
          # of object page table entries   = 00000002H
          reserved                         = 00000000H
          flags = READABLE|WRITABLE|BIG

object 3: virtual memory size              = 00004000H
          relocation base address          = 00120000H
          object flag bits                 = 00002003H
          object page table index          = 00000010H
          # of object page table entries   = 00000000H
          reserved                         = 00000000H
          flags = READABLE|WRITABLE|BIG
...
```

Figure 3.3: Example showing the Object table for the LPSK kernel.

To retrieve the starting stack address the `wdump` tool can be leveraged. The `wdump` tool analyzes a chosen executable file and provides information about its structure. The object table for the LPSK kernel binary has three entries; code, data, and stack. As mentioned in Section 3.3.1 the GRUB bootloader respects the address specified in the `lpsk_kernel` binary. In Figure 3.3 the third object is the stack object. The relocation base address field specifies the linear address for the stack, which is 0x00120000. This value added to the stack pointer at the beginning of a function call is the end of the parameter list.

The linear address to probe is that of the `kmem_allocate` function. This can be retrieved from the MAP file generated during compilation. Figure 3.1 shows the address for the function `kmem_allocate`. However, since the allocated linear address is written to the `start_addr` parameter during the execution of the function, the starting address of the function cannot be used. To ensure that the parameter is read after it has been written to with the correct value, the

address of the return statement for the function will be used. A simple way of retrieving this address is to get the start address of the next function in the MAP file and subtract the size of the previous functions return statement. The size of return statements in the LPSK is 3 bytes In Figure 3.1 the next function is `kmem_free` which is at offset 0x0000b9d1. By subtracting 3 bytes, the offset becomes 0x0000b9ce. However, the relocation base address for the code segment must be added to this offset to derive a linear address. In Figure 3.3 the relocation base address for the code segment is 0x0010000, thus the address the probe needs to be installed on is 0x0010b9ce.

A final aspect to using the VProbe is displaying the name of the module that is being loaded into memory. The name of the module is the ascii file name at the time the bootloader read it into memory. This is not a requirement to build the symbol table, but it makes it easier for the end user. If the VProbe does not display the name of the binary being loaded then the configuration vector must be examined to see which binary is being loaded. As mentioned in Section 3.3.1 the `find_module` function will be called for every binary loaded into memory. The third parameter is a path to the binary that GRUB loaded from disk, which will contain the name. Another VProbe can be set on the address of `find_module` to retrieve the path parameter. This will allow the name of each binary to be displayed before the linear addresses of its segments are displayed. However, this requires a third piece of information to be encoded into the VP script: the linear address of the `find_module` function.

3.3.3 VProbe for Symbol Table Creation

Section 3.3.2 laid out the guidance for creating a VProbe to obtain linear starting addresses for the segments of each binary loaded into memory. This section will present the source code for that VProbe. The source code in its entirety can be seen in Figure D.4 of Appendix D.

The VProbe listed in Figure D.4 has two user defined functions; the `getArg32` function, and `derefTcxKernelStack` function. The `getArg32` function retrieves an argument off the stack based on the offset and the argument number. It is important to note that this function assumes a parameter is 4 bytes wide. Parameters can be larger than 4 bytes so the calling code must account for this. The `derefTcxKernelStack` function returns the first 4 bytes of data pointed to by the pointer parameter. The pointer value will have the linear stack base address added to it before dereferencing the pointer.

```
>vmrun vprobeLoad /path/to/vmxfile.vmx "`cat vpscript`"
>vmrun vprobeListProbes /path/to/vmxfile.vmx
>vmrun vprobeListGlobals /path/to/vmxfile.vmx
>vmrun vprobeReset /path/to/vmxfile.vmx
>vmrun vprobeVersion
```

Figure 3.4: Example loading a VProbe with the vmrun command[11].

The above user defined functions provide the necessary functionality to build the VProbe discussed in Section 3.3.2. In fact, the code listed in Figure D.4 installs two separate probes. One probe monitors the `find_module` function and prints the loaded module name. The second monitors the `kmem_allocate` function and displays the linear base address that is allocated for a given segment.

3.3.4 Adding VProbes

Once the VProbe for harvesting the linear addresses has been encoded with all the appropriate data, the probe must be loaded into the Virtual Machine Monitor. This is done by the `vmrun` command. The `vmrun` command is part of the VMware Workstation product. The `vmrun` tool has many other uses. Only the portions pertaining to VProbes will be covered in this section.

Figure 3.4 shows example uses of the `vmrun` command for VProbes. The top command in the figure provides example usage for loading a VProbe. The virtual machine that the vprobe is installed on is specified by the vmx file. The source code for the VProbe is passed on the command line as a parameter to the `vmrun` tool. The syntax used in Figure 3.4 is specific to bash shell. VMware recommends using this syntax and encourages Windows users to leverage Cygwin when using `vmrun` and VProbes. Furthermore, the VProbe Generator GUI, a tool developed for this thesis, can add VProbes to a running virtual machine. See Appendix C for details on the VProbe Generator GUI.

Note that VProbes can only be added to the Virtual Machine Monitor once the virtual machine is in a running state. In order to generate a symbol table the VProbe in Figure D.4 must observe the kernel's loader during boot. This implies that the VProbe must be installed before the kernel has been loaded. A logical place during execution for this VProbe to be loaded is when the bootloader is executing. Note that on virtual machine power down all probes are released and no longer present. Thus a VProbe must be added every time a Virtual Machine is turned on.

```
....
Find_modules exe name = /scss
kmem_allocate load_addr = 0x14b000
kmem_allocate load_addr = 0x14f000
kmem_allocate load_addr = 0x157000
Find_modules exe name = /pl1.gts
Find_modules exe name = /scos
kmem_allocate load_addr = 0x15b000
kmem_allocate load_addr = 0x15f000
kmem_allocate load_addr = 0x161000
Find_modules exe name = /pl2.gts
Find_modules exe name = /tsm_appl
kmem_allocate load_addr = 0x165000
kmem_allocate load_addr = 0x169000
kmem_allocate load_addr = 0x16b000
....
```

Figure 3.5: Example vprobe.out contents when script in Figure D.4 is executed.

3.3.5 Interpreting Output

After a VProbe is loaded all output is sent to the `vprobe.out` file. By default, this file is placed in the same directory as the `vmx` file. The contents of the `vprobe.out` file is controlled by the output operations in the VProbe.

Figure 3.5 shows the output produced when the script in Figure D.4 is executed. The print statements in the VProbe correspond to the contents of Figure 3.5. In Figure 3.5 there are multiple calls to `kmem_allocate` for each module. Each of these these calls corresponds to a segment within that module. Typically the segments are arranged as follows: code, data, and stack. For example, the `SCOS` module code segment begins at address 0x15b000. The VProbe Generator GUI provides an application to assist in viewing this file in real time. See Appendix C for details.

3.3.6 Final Symbol Table

The final symbol table can be built one MAP file at a time. The general algorithm for building the symbol table is as follows: Take the linear address generated by the harvester.vp script and add it to the offset of the symbol name in the MAP file. However, the result must be in a format that is understandable by the VMware Workstation product.

```
...
100028     T     __halt0
10002b     T     _asm_init_video0
100057     T     __asm_put_char0
100093     T     _get_ax0
100094     T     _get_bx0
100097     T     _get_cx0
10009a     T     _get_dx0
10009d     T     _get_ss0
1000a0     T     _get_cs0
...
10a7b7     T     _kio_printf0
10a7dd     T     _kio_printf_str0
10a80e     T     _kio_printf_int0
10a838     T     _kio_printf_char0
10a863     T     _debug_putchar0
10a962     T     _printf_bottom_line30
10aa23     T     _printf_bottom_line10
10aab0     T     _test_function0
10aae9     T     _itoa0
10ac8d     T     _strncpy0
10ad05     T     _strncat0
10adb2     T     _memcpy0
10adf5     T     _memcmp0
10bf89     T     _lpsk_powerdown0
...
```

Figure 3.6: Example output of LPSK Kernel Binary symbol table.

By default, VMware Workstation correctly parses output of the Linux kernel symbol table [11]. This output is generated by the `nm` utility[3]. This format was chosen because the `nm` utility is open source and a large amount of documentation on it's output format exists.

Figure 3.6 shows the `nm` format for the LPSK kernel binary. The output has three columns; address, type, and name. Address is the linear address in memory where the symbol resides. The type denotes the form of the symbol. The form specifies whether a chosen symbol is a variable or function. For example, the `T` type denotes that the symbol is in the code segment. The third column contains the name, the mnemonic which can be used in VProbes instead of the linear address. Multiple tools have been developed for this thesis for assisting in generating symbol tables for the LPSK. See Appendix B and C for a description of these tools.

The TCX kernel has the ability to load the same executable into different partitions. Thus, the same symbol name can exist in multiple places in memory. The Symbol Generator tool in

```
vprobe.guestSyms = "symbol.sym"
```

Figure 3.7: Configuration option for adding symbol files to vmx files [11].

Appendix C accounts for this by applying the partition number to the end of the symbol. The symbols in Figure 3.6 are in partition 0 and each symbol has a 0 concatenated to the end of it's name.

3.4 Using the Symbol Table

For the symbols to be available inside of VP scripts the Virtual Machine Monitor must know the location of the symbol table. This is done by specifying the symbol table in the Virtual Machine configuration file [11]. This thesis adopts VMware's convention of using a `.sym` extension [11] for symbol table files.

Figure 3.7 shows the configuration option for adding a symbol file to a `vmx` file. If this file is added while the virtual machine is running it will not be available until the Virtual Machine Monitor has been restarted. This implies that the virtual machine will have to be run once to generate the symbol table and then restarted to make those symbols available for use in VP scripts.

3.5 Verification Tests

To ensure that the use of VProbes is displaying the correct state of the LPSK, a test bed has been developed. This is not a definitive proof of the correctness of VProbes, but a simple test to verify the correctness of the approach. Two verification tests have been designed: the vertical test and the horizontal test. These two tests use unique aspects of the kernel's functionality that are not present on other operating systems. As such, these tests are designed to exercise these functionalities and ensure that VProbes retrieves the correct values.

The vertical test is designed to exercise VProbes' ability to follow code execution in multiple hardware privilege levels. Most operating systems run in privilege level (PL) 0 and their applications execute in PL3. The LPSK uses all four of the Intel hardware privilege levels. The vertical test will execute code in all four hardware privilege levels while a VProbe attempts to retrieve information about this execution.

```
int test_scos_gate( int val );
int scos_test( int val );
int test_scss_gate( int val );
int scss_test( int val );
int test_lpsk_gate( int val );
int lpsk_test( int val );
```

Figure 3.8: List of function calls in vertical test.

The horizontal test is designed to test VProbes ability to handle multiple data segments. The multi-segmented nature of the LPSK allows for multiple data segments to exist at each privilege level. This test will assign known values into two additional data segments, one at privilege level 1 and the other at PL0. A VProbe will be used to retrieve information about this write and the results will be compared to the known value.

3.5.1 Vertical Test Definition

The vertical test watched execution flow of a function call from the TSM Application down to a kernel function. This required the TSM Application to first perform a gate call to SCOS executing in PL2. SCOS then performs a gate call into SCSS executing in PL1. Finally, SCSS made a gate call into the LPSK kernel.

Figure 3.8 shows the function calls in the VProbe test. The TSM Application first calls `test_scos_gate`, which is a gate call into hardware privilege level 2. The gate call then executes the `scos_test` function in the SCOS binary. The `scos_test`, in turn, calls the `test_scss_gate` gate call to enter into privilege level 1. Then the gate call executes the `scss_test` function in the SCSS binary. Finally, the `test_lpsk_gate` gate call is made, thus moving execution to privilege level 0. In turn, the `lpsk_test` function is executed.

Each of the functions takes an integer as a parameter. This parameter is left unchanged by all of the functions and is passed down to the kernel. Thus the parameter for all six function calls will be the same value. An execution VProbe can be set to retrieve the first parameter for each function when it begins executing, This parameter is retrieved from the stack memory segment in each privilege level. All of these values retrieved from the VProbe should be the same and match the value the TSM Application provided. The verification test will call the `test_scos_gate` with a parameter value of 1.

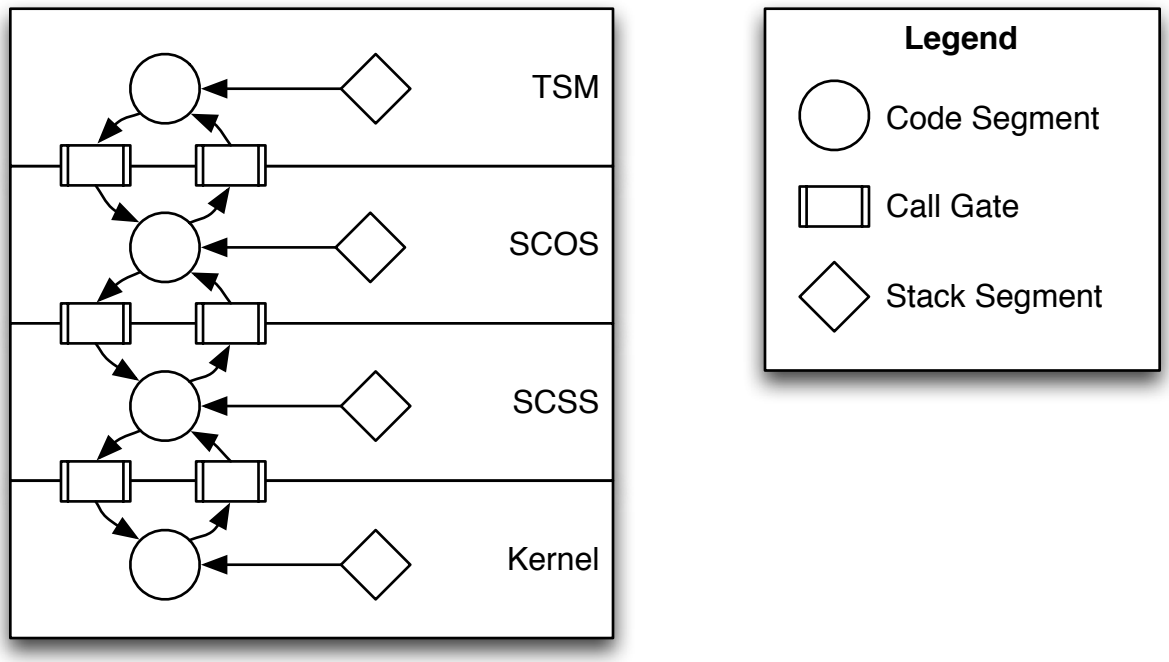

Figure 3.9: Diagram showing the execution flow of the vertical test.

Once the execution flow has reached the `lpsk_test` function the parameter will have 1 added to it and the resultant value is returned. All of the functions in Figure 3.8 have an integer return value and an execution VProbe can be set on the return statement in each function. Convention dictates that return values are placed into the EAX register before returning from function calls. This implies that the execution VProbe must retrieve the EAX register when the return statement is executed. Each value retrieved using these VProbes should be equal to the parameter value plus one. In the case of this test, the value will be equal to 2. Figure 3.9 shows the execution flow of the vertical test.

If all these values match the expected values, then the VProbe has in fact retrieved the correct information.

3.5.2 Vertical Test Construction

To build this test the LPSK source code must be modified to add the functions listed in Figure 3.8. The TSM Application only needs to call the `test_scos_gate` gate call. Thus all the other code within the TSM Application can be removed. To help in instrumenting the call, the application will first require a user to hit enter before calling the appropriate gate call. This instrumentation will allow the user to add the test VProbe(s) before the first function call is made.

This test will require that a total of 12 VProbes be installed. Six of the probes will be installed on the entry of each function listed in Figure 3.8. These probes will retrieve the parameter value passed to each function. The parameter values are read from the stack segment in each privilege level. Each of these functions will be referred to by name listed in the symbol table in Figure 3.8. Details on constructing symbol tables for the LPSK can be found in Section 3.3.

The remaining six VProbes will be installed on the return statement of each of the functions in Figure 3.8. The purpose of this VProbe is to retrieve the value returned from the `lpsk_test`. Since symbol table only contains the starting address for functions, the linear address for the function's return statement will be used. A discussion of determining return statement linear addresses can be found in Section 3.3.2. Figure D.1 displays the source code for the test VProbe.

```
...
test_scos_gate( 0x1 )
scos_test( 0x1 )
test_scss_gate( 0x1 )
scss_test( 0x1 )
test_lpsk_gate( 0x1 )
lpsk_test( 0x1 )
lpsk_test = 0x2
test_lpsk_gates = 0x2
scss_test = 0x2
test_scss_gate = 0x2
scos_test = 0x2
test_scos_gates = 0x2
...
```

Figure 3.10: Output from vertical test VProbe.

3.5.3 Vertical Test Execution

Figure 3.10 displays the output from Figure D.1. Note that each parameter has a value of one and each return has a value of two. These values match the expected results mentioned in the test definition. Thus we can be assured that the VMware VProbes mechanism is retrieving the correct information from the target system.

3.5.4 Horizontal Test Definition

The horizontal test ensures that the VProbe mechanism can handle the multi-segmented nature of the LPSK. The name horizontal comes from the ability of the LPSK to have multiple code and data segments in a chosen privilege level.

This test used a data VProbe to watch variables in a second data segment in both privilege levels 1 and 0. This equates to the SCSS module and the LPSK kernel locations. However, the test will be started by the TSM application in privilege level 3. The call stack from Figure 3.8 will be identical in this test. The operation of the TSM application and SCOS functions will not change.

The `scss_test` will first read the `first_segment_variable_scss` and assign it to the variable in the second data segment. Then the function will assign the value of the parameter to a variable located in the second data segment. This function will subsequently make a gate call into the LPSK with the `test_lpsk_gate` function. As in Section 3.5.1 the parameter value

```
int first_segment_variable_scss = 0;
int scss_test( int val )
{
        second_segment_variable_scss = first_segment_variable_scss;
        second_segment_variable_scss = val;
        return test_lpsk_gate( val );
}
int first_segment_variable_lpsk = 0;
int lpsk_test( int val )
{
        second_segment_variable_lpsk = first_segment_variable_lpsk;
        second_segment_variable_lpsk = val;
        return val + 1;
}
```

Figure 3.11: Example code for the SCSS and LPSK horizontal test.

passed into this function will be left unchanged and is provided as a parameter to the gate call. Example code for the `scss_test` function can be found in Figure 3.11. Notice in Figure 3.11 that the second data segment variable is first set to zero and then the parameter value.

The `lpsk_test` will behave in a similar fashion to the `scss_test` function. The `first_segment_variable_lpsk` will be read and assigned to the variable in the second segment. Then the parameter value will be assigned to the variable in the second data segment. However, no further gate calls can be made as execution resides in the lowest privilege level. Thus, as in Section 3.5.1, the kernel test function will return the parameter value plus one. Example code for the `lpsk_test` function can be found in Figure 3.11. The execution flow of the horizontal test is presented in Figure 3.12.

The parameter value will be set to one in the TSM application. This test will watch the following variables: `second_segment_variable_scss`, `second_segment_variable_lpsk`, `first_segment_variable_scss`, and `first_segment_variable_lpsk`. When the execution flow of the system reads from `first_segment_variable_scss` and `first_segment_variable_lpsk` a VProbe will execute, printing to the `vprobe.out` file “Read variable name”. When execution flow writes values to `second_seg ment_variable_scss` and `second_segment_variable_lpsk`, the VProbe will execute and retrieve the values written. If VProbes retrieves a value of 1 then the VProbe has correctly handled multi-segments within a given privilege level.

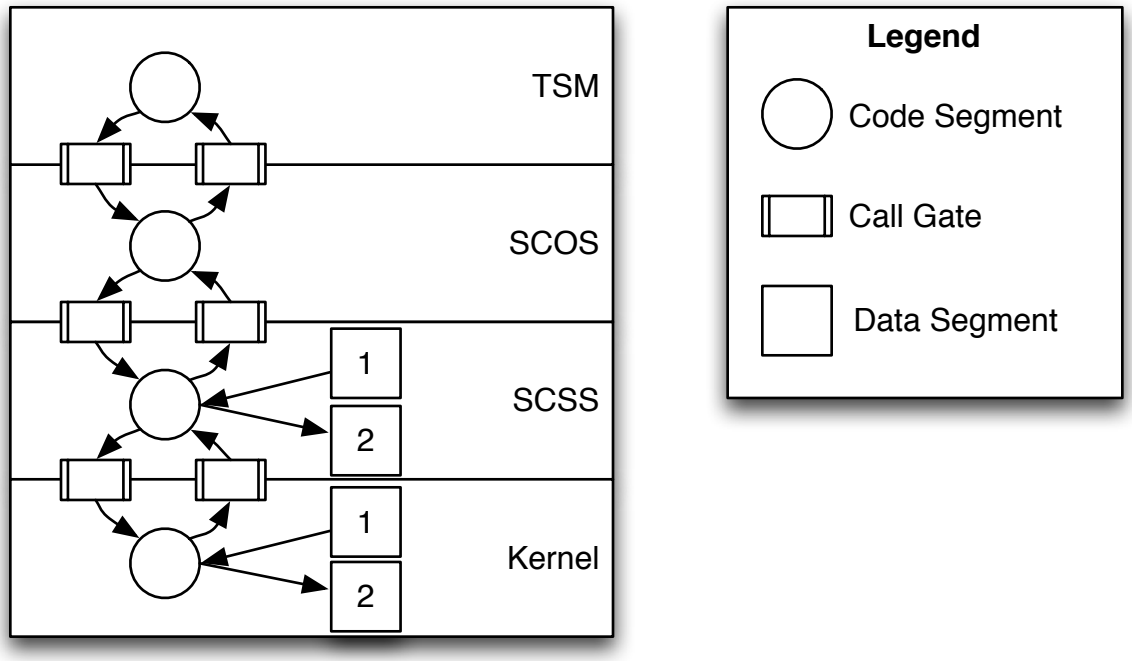

Figure 3.12: Diagram showing the execution flow of the horizontal test.

3.5.5 Horizontal Test Construction

To build this test the LPSK source code must be modified. The modified code is similar to the modifications presented in Section 3.5.2. However, the `scss_test` and `lpsk_test` functions are modified to initialize the variables in the second data segment with variables from the first data segment. Then the parameter value will be assigned to the variable in the second data segment. This variable resides in a second second data segment for each privilege level. These modifications can be seen in Figure 3.11.

This test requires four data VProbes. Two VProbes will watch the memory locations for the variables `first_segment_variable_scss` and `first_segment_variable_lpsk`. These Vprobes will execute when these variables are read. The remaining two probes will watch the memory locations for the variables `second_segment_variable_lpsk` and `second_segment_variable_scss`. When these variables are written to, the VProbe will execute and retrieve the value written to the memory locations. The source code for the horizontal test VProbe be found in Figure D.2.

```
Read first_segment_variable_scss
_second_segment_variable_scss = 0x0
_second_segment_variable_scss = 0x1
Read first_segment_variable_lpsk
_second_segment_variable_lpsk = 0x0
_second_segment_variable_lpsk = 0x1
```

Figure 3.13: Output from Horizontal Test VProbe.

3.5.6 Horizontal Test Execution

Figure 3.13 presents the output from the horizontal test VProbe. Note that each line represents a write to the variable. The first write to each variable is 0. This write ensures that the memory location in the second segment does not start with a value of 1. The second write to each variable is 1. Since the values match our expected value of 1, we can be assured that VProbes has handled multiple data segments correctly.

3.6 Other Experiments

This section contains experiments not directly related to verifying that VProbes works on the LPSK. These tests were run to understand the operating procedures of VProbes as well as the Intel architecture. As such, it presents results that will help readers understand the use of VProbes.

3.6.1 Write to Read Only Segment

As discussed in Section 2.6, the LPSK makes heavy use of segmentation. With the large number of different memory segments it becomes a reasonable possibility that code execution may violate permissions for a given segment. This test attempted to write to a segment marked as read only. The goal was to ascertain if the write succeeded before an interrupt is generated notifying the system of an access violation.

This test was developed in the SCSS module. A write to the first 4 bytes of the keyboard mapping segment was attempted. The keybaord mapping segment is marked in the configuration vector as read only. Two VProbes were used to observe the execution of the test. The first VProbe watched the first byte of the read only memory segment for writes. The second VProbe watched the execution of the interupt 13 service handler. The source code for this VProbe can be found in Section D.3 from Appendix D.

When the test was executed the interupt 13 service handler fired before writing to the read-only memory segment. Thus no writing to the read only memory segment succeeded.

3.7 Summary

This chapter presented building symbol tables for the LPSK and using them to instrument VMware VProbes. Furthermore, two verification tests were presented to verify the output generated by VProbes. The next chapter will present related work for systemic debugging for operating systems.

CHAPTER 4:
Related Work

This chapter presents other methodologies that are being used to debug operating systems.

4.1 DTrace

DTrace [4] is a dynamic instrumentation tool that VMware VProbes was modeled after [11]. Currently DTrace is implemented for Solaris and FreeBSD. DTrace is an instrumentation tool that helps in debugging problems in operating systems. Like GDB, it must be installed in the kernel for it to function.

DTrace implements its own language referred to as D [4]. The structure of the D language allows users to specify conditions. These conditions are known as probes [4]. When the condition of the probe becomes true then the defined action executes [4]. A given probe will execute every time the condition is true.

DTrace requires that code must register potential probe points. That is, each possible place in the target code must have code added to allow DTrace to instrument it. These instrumentation points are referred to as providers [4].

A DTrace kernel module must be present on the target system. This brokers comminucation between the DTrace probes and the running kernel. This approach is similar to GDB that requires a server to run in kernel space. However, unlike GDB, DTrace will incur no overhead when probes are not being run.

4.2 K42

K42 is a research kernel developed by IBM [1]. The overall goal being to re-vitalize dated operating system engineering principles. The main goal of the K42 research group was to develop a high performance operating system for modern processors [1]. The development effort aims to adhere to the Linux API thus requiring minimal porting of application code [1].

From the outset of this project the developers had debugging and system performance testing in mind. All kernel code has debugging routines present and debugging may be dynamically enabled [2]. Furthermore, the collection of debugging events is separated from their analysis

[2]. This implies that instrumented kernel routines will only collect and distribute debugging information. The client applications that harvest the debugging information will have to analyze the input.

This implies that kernel code does not need to be compiled into a debugging mode [2]. These debugging routines can be enabled and accessed from user space [2]. There is no cost for these instrumentation points when the debugging routines are not being used [2]. This artifact is similar to DTrace.

4.3 In-Circuit Emulation

In-Circuit Emulation (ICE) requires the use of a hardware device that helps developers debug low level problems [6]. Traditionally, the ICE device is inserted into the processor socket [6]. This allows the ICE device to act as the processor and completely control the target system.

ICE devices allow functionalities like breakpoints without affecting the systems state [6]. This is possible because the device replaces system hardware, thus it is operating at the lowest possible layer. An ICE also allows for extensive program tracing [9]. The ICE device allows real time dumps of executing code for the developer to scrutinize.

The biggest advantage of ICE is there is no effect on the target system [6]. The non-intrusive nature of ICE makes it an extremely attractive technology. However, ICE devices are not general purpose. Specifically, each ICE device must match to a given make, model and revision of target processor [9]. Thus, for every hardware platform upon which the code is to execute a specific ICE device must be purchased.

4.4 GDB

As mentioned in Chapter 2, GDB is used to debug operating systems, such as Linux and FreeBSD. This is done by compiling a GDB kernel module that acts as server [7]. This module, executes alongside the kernel and has the ability to inspect the operating system. This approach gives developers the ability to set breakpoints, step instructions, and trace programs all within the kernel.

While this approach provides developers with a window into the running kernel, it is not a perfect solution. The act of running the debugger inside of the kernel has the inherent defect of changing system state. Specifically, the debugger can change the state of the program it is

attempting to observe. This is a different approach than that of ICE and VProbes which are non-intrusive in nature.

4.5 Summary

This chapter presented related work for debugging technologies. The next chapter presents conclusions and future work for the application of VMware VProbes to the LPSK.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion and Future Work

This chapter concludes this thesis and presents future work related to the application of VMware VProbes to a segmentation based separation kernel.

5.1 Conclusion

This thesis set out to determine if VMware Vprobes could be used to debug the LPSK. Background information was gathered to understand topics such as virtualization, symbol tables, Open Watcom build tools, and debugging technologies. This provided a basis of comparison for VMware VProbes. Using the background information a methodology was presented for using VMware VProbes on the LPSK. Furthermore, attention was paid to building symbol tables so that developers could refer to addresses by name.

The current state of the LPSK is constantly changing. As such, the VP scripts used in this thesis may become obsolete. However, the methodology they use will remain viable so long as VMware VProbes functional specifications do not change.

5.2 Future Work

This section presents future work based upon the efforts in this thesis.

5.2.1 VMware VProbe Generator GUI

The current version of the VMware VProbe generator GUI provides an example of how to construct a tool to assist in the application of VProbes. To properly develop a fully functional application, requirements must be gathered. Engineering effort should be aimed at identifying the functional requirements. Specifically, the current version is aimed at solving problems that were of immediate concern. Future versions should address the needs of LPSK developers from the perspective of usability.

5.2.2 Automation of Linear Address Mapping

After the work in this thesis was completed, it became obvious that it is possible to automate the process described in Appendix A. The output from the harvester.vp script can be parsed and

the information applied to the MAP files. This automation would provide developers with a tool that would perform all of the tasks in Appendix A. Thus, effectively reducing the complexity of the tool and enhancing its usefulness.

5.2.3 Library of VP Scripts

As developers begin to use VMware VProbes, many scripts will be developed for debugging certain aspects of the kernel. To reduce redundant work, it will be useful to archive these scripts to increase reuse. Furthermore, a general purpose mechanism for writing VP scripts could be devised. This mechanism could take VP scripts written for a specific purpose and generalize them. This effectively makes specific scripts useful in a wide variety of situations.

List of References

[1] Jonathan Appavoo, Marc Auslander, Dilma DeSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jim Xendix. *K42*. IBM, August 2002. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/k42.index.html`.

[2] Jonathan Appavoo, Marc Auslander, Dilma DeSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jim Xendix. *K42's Performance Monitoring and Tracing*. IBM, November 2002. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/k42.index.html`.

[3] GNU Binutils. *GNU NM Manual*. `http://sourceware.org/binutils/docs-2.19/binutils/nm.html#nm`.

[4] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. *Usenix*, 2004.

[5] Paul C. Clark, Cynthia E. Irvine, Thuy D. Nguyen, Timothy E. Levin, Timothy M. Vidas, and David J. Shifflett. *SecureCore Software Architecture: SecureCore Operating System (SCOS) Functional Specification*. CISR, 1411 Cunningham Rd, Monterey, CA 93943 USA, 2007. NPS-CS-07-018.

[6] Jack Ganssle. Beginner's corner - in-circuit emulation. *Embedded Systems Design Magazine*.

[7] John Gilmore and Stan Shebs. *GDB Internals*, Bebruary 2004. `http://www.gnu.org/software/gdb/documentation/`.

[8] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112. ACM, New York, NY, USA, 1973.

[9] Jonathan Hector. How to choose an in-cicuit emulator. *Embedded.com*, 2002.

[10] IBM. Ibm os/2 16/32-bit object model format (omf) and linear executable module format, October 1996. `http://www.openwatcom.org/ftp/devel/docs/lxomf.pdf`. [Online; accessed June-11-2009].

[11] VMware Inc. *VProbes Programming Reference*. VMware Inc., 3401 Hillview Ave. Palo Alto, CA 94304 USA, 2008.

[12] Cynthia E. Irvine, Timothy E. Levin, Thuy D. Nguyen, and George W. Dinolt. The trusted computing exemplar project. In *Proceedings of the 5th IEEE Systems, Man and Cybernetics Information Assurance Workshop*. IEEE, 2004.

[13] John R. Levine. *Linkers and Loaders*. Morgan-Kauffman, 1999.

[14] Mark A. Linton. The evolution of dbx. In *Proceedings of the Usenix Summer 1990 Technical Conference*, June 1990.

[15] Gordon Matzigkeit and Yoshninori K. Okuji. *The GNU GRUB Manual*. GNU, 51 Franklin Street, Fifth Floor, Boston, MA 02111, USA, 2006.

[16] Thuy Nguyen, Timothy E. Levin, and Cynthia E. Irvine. Tcx project: High assurance for secure embedded systems. In *11th IEEE Real-Time and Embedded Technology and Applications Symposium Work-In-Progress Session*. IEEE, 2005.

[17] Yoshinori K. Okuji, Bryan Ford, Erich Stefan Boleyn, and Kunihiro Ishiguro. *The Multiboot Specification*. GNU, 51 Franklin Street, Fifth Floor, Boston, MA 02111, USA, 2006.

[18] Open Watcom Project. *Getting Started*, 2006. `http://www.openwatcom.org/index.php/Manuals`.

[19] Open Watcom Project. *Open Watcom Linker User's Guide*, 2006. `http://www.openwatcom.org/index.php/Manuals`.

[20] QEMU Project. *QEMU Internals*, February 2007. `http://bellard.org/qemu/qemu-tech.html`.

[21] David J. Shifflett. *Build and Test Instructions for Least Privilege Separation Kernel (revision 85)*. CISR, January 2009.

[22] David J. Shifflett, Paul C. Clark, Cynthia E. Irvine, Thuy D. Nguyen, Timothy M. Vidas, and Timothy E. Levin. *SecureCore Software Architecture: Trusted Management Layer (TML) Kernel Extension Module Interface Specification*. CISR, 1411 Cunningham Rd, Monterey, CA 93943 USA, 2008. NPS-CS-07-021.

[23] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Xplore*, 2005.

[24] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB*, February 2004. `http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html`.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Manual for TCX Debugging with VProbes

This Appendix presents a step by step guide to enabling VProbes and building a symbol table for the LPSK. All directions will be given for a Linux host.

A.1 Prerequisites

The following assumptions will be made.

- VMware Workstation 6.5 is installed on the Linux host and is being used as the Virtual Machine Monitor.
- It is assumed that the LPSK Virtual Machine has been set up [21]. Furthermore, the developer has access to all files created during the build process. For the purposes of building symbol tables, the user needs all files with a `.map` extension and all executable files including the `lpsk_kernel` binary.
- This guide will refer to the directory in which build files are located as `BUILD_DIR`. This directory is located on the user's build machine [21].
- It is assumed Vprobes has been enabled for the VMware virtual machine. For directions on enabling VProbes for VMware Workstation see [11].
- The directory used for scratch space when building a symbol table will be referred to as `SYM_DIR`. This directory should be located in the file system of the host that is running VMware Workstation.
- The `harvester.vp` script is located in the `SYM_DIR`.

A.2 Building Symbol Tables

1. Move all MAP files (file with a `.map` extension) from `BUILD_DIR` to `SYM_DIR`. The file named `lpsk_kernel` should be moved from `BUILD_DIR` to `SYM_DIR`.
2. Locate the address of the `find_module` and `kmem_allocate` functions. In the Memory Map section of the `lpsk_kernel.map` file, Figure A.1 shows an example

```
...
0001:0000b4bc  _kmem_preallocate
0001:0000b6fa  _kmem_allocate
0001:0000b9d1+ _kmem_free
...
0001:000057ad+ _create_gates
0001:00005ace  _find_module
0001:00005bbd  _load_partition
...
```

Figure A.1: Example excerpt from the LPSK kernel MAP file.

excerpt from the `lpsk_kernel.map`. The address for the `kmem_allocate` function is 0x0000b6fa. The address for the `find_module` function is at address 0x00005ace.

3. Apply the linear offset address to the `find_module` and `kmem_allocate` function addresses. To obtain the linear offset address the `wdump` tool can be leveraged. Figure A.2 shows the output from running `wdump` on the lpsk_kernel binary. Notice that Object 1 is the code segment and the relocation base address is 0x00100000. The relocation base address can then be added to the addresses harvested in Step 2. Thus, `find_module` address becomes 0x00105ace and `kmem_allocate` becomes 0x0010b6fa.

4. Locate the return address for the `kmem_allocate` function. This address is located by subtracting 3 from the address of the next function in the `lpsk_kernel.map`. In Figure A.1 the next function is `kmem_free`. The address of the `kmem_free` function is 0x0010b9d1. Thus the address of the `kmem_allocate` function return statement is 0x0010b9ce.

5. The string `FINDMOD` is used in the harvester.vp file as a place holder. This string should be replaced with the address of the `find_module` function. This address was retrieved in Step 2.

6. The string `KMEMALLOC` is used in the harvester.vp file as a place holder. This string should be replaced with the return address of the `kmem_allocate` function. This address was retrieved in Step 4.

7. Locate the stack base address of the LPSK kernel. Use the wdump tool to analyze the `lpsk_kernel` in the `SYM_DIR`. Figure A.2 shows an example of running the wdump tool. The object table output shows that the `lpsk_kernel` has 3 objects. The code, data and stack segments, respectively. The third object is the kernel's stack. The stack

```
#wdump lpsk_kernel
...
                                  Object Table
==========================================================================
object 1: virtual memory size              = 0000C21FH
          relocation base address          = 00100000H
          object flag bits                 = 00002005H
          object page table index          = 00000001H
          # of object page table entries   = 0000000DH
          reserved                         = 00000000H
          flags = READABLE|EXECUTABLE|BIG

object 2: virtual memory size              = 0000E8D4H
          relocation base address          = 00110000H
          object flag bits                 = 00002003H
          object page table index          = 0000000EH
          # of object page table entries   = 00000002H
          reserved                         = 00000000H
          flags = READABLE|WRITABLE|BIG

object 3: virtual memory size              = 00004000H
          relocation base address          = 00120000H
          object flag bits                 = 00002003H
          object page table index          = 00000010H
          # of object page table entries   = 00000000H
          reserved                         = 00000000H
          flags = READABLE|WRITABLE|BIG
...
```

Figure A.2: Example usage of the wdump tool on the LPSK kernel binary.

base address for the LPSK kernel is the relocation base address, 0x00120000 in Figure A.2

8. Replace the `STACKADDR` string in the harvester.vp file with the hex representation of the stack base address.

9. On the machine running VMware Workstation remove the `vprobe.out` file if it exists. This file will reside in the same directory as the vmx file. If VProbes have never been run on TCX virtual machine then, it will not exist.

10. Power on the TCX Virtual Machine. Once the GRUB boot loader initializes press the `ESC` key. When GRUB initialization occurs, the screen will display `GRUB Loading stage1.5`. This will cause the GRUB boot loader to pause until the user selects a boot option.

```
>vmrun vprobeLoad /path/to/tcx.vmx "`cat harvester.vp`"
```

Figure A.3: Command showing how to load the harvester.vp file with the vmrun command.

```
vprobe.guestSyms = "/SYM_DIR/TCX.sym"
```

Figure A.4: Configuration option for adding symbol files to vmx files [11].

```
>vmrun vprobeListProbes /path/to/tcx.vmx
```

Figure A.5: Example usage of the vmrun utility to list symbols.

11. Open a command shell on the host system running VMware Workstation. Load the harvester.vp script with the vmrun utility. Figure A.3 shows the command line instrumentation and parameters needed to load the harvester.vp file with the vmrun utility (This assumes that harvester.vp has been placed in `SYM_DIR` and that the command line session has a current working directory of `SYM_DIR`) The `/path/to/tcx.vmx` is the absolute file name of the vmx file for the TCX virtual machine.

12. Return to the TCX virtual machine, which is paused waiting for user input. A list of operating systems will be showing. Select the TCX boot option from the GRUB boot menu and let the virtual machine boot.

13. After the TCX virtual machine has completely finished booting, copy the `vprobe.out` file to the `SYM_DIR`. This file will be located in the same directory as the `vmx` file, and will be created when the VProbe runs.

14. Use the Symbol Table Generator from the VProbe Generator GUI or the TCX Symbol Generator to build the symbol table. For directions on these see Section C.2 or Section B.1. Save the symbol file as `TCX.sym` in the `SYM_DIR`.

15. Add the symbol table file to the TCX vmx file. Figure A.4 shows the configuration option to add to the vmx file. Note that `"TCX.sym"` must be given as a full absolute path to the TCX.sym file in the `SYM_DIR`.

16. Reboot the TCX virtual machine. Once the TCX virtual machine has been restarted all symbols listed in the TCX.sym file will be available to VProbes. To ensure that the symbols have been loaded correctly the vmrun tool can be used. Figure A.5 shows how to use vmrun to display the symbols. The output of A.5 will contain the symbols listed

in the TCX.sym file. The vprobeListProbes command will retrieve all available probe points.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
TCX Symbol Generator

This chapter contains a brief description of the TCX Symbol Generator tool. This is a command line utility that applies an address offset to a MAP file and outputs an NM style symbol table. This tool was written for the Linux operating system using GNU's libc.

Figure B.1 displays the command line options for the tool. All options listed must be set for the tool to execute. The `-m` option specifies the name of the MAP to parse. The `-a` option specifies the linear address offset to apply to each symbol in the map. The `-o` option allows the user to specify the name of the output file. The output file will be appended to, if it already exists. If the output file does not exist one will be created. The `-s` option specifies which memory segment in the map file to pull symbols from.

B.1 Building Symbol Tables

It is assumed that a user reading this section has access to all the MAP files created during the compilation process. Furthermore, it is assumed that the `harvester.vp` script has been run and the resultant vprobe.out file is available.

Figure B.2 shows usage of the TCX Symbol Generator for building a symbol table. The example shows how to apply a linear base address offset to the SCOS map file. Figure B.2 has a linear address offset of `0x0015b000` and only the symbols in segment one of the SCOS MAP file are being added to the `tcx.sym` file.

Figure B.3 shows the output from the harvester.vp script. Using this output a symbol table for TCX can be built. For example, to create the symbol table for the SCOS module the command in Figure B.2. By looking at Figure B.3 the first segment in SCOS is at address 0x0015b000.

```
./ tcx_symbol_generator -m filename -a 0x00000000 -o filename -s x
Options
-m filename:    Map file to apply the fixups to.
-a 0x0000000    32bit address offset (in hex)
-o filename:    The place to write the symbol table to.
-s x        :   The code segment to build symbols for
```

Figure B.1: Usag of the TCX Symbol Table.

```
./tcx_symbol_generator -m scos.map -a 0x0015b000 -o tcx.sym -s 1
```

Figure B.2: Example parameters for symbol table generation.

```
....
Find_modules exe name = /scss
kmem_allocate load_addr = 0x14b000, Object # = 1
kmem_allocate load_addr = 0x14f000, Object # = 2
kmem_allocate load_addr = 0x157000, Object # = 3
Find_modules exe name = /pl1.gts
Find_modules exe name = /scos
kmem_allocate load_addr = 0x15b000, Object # = 1
kmem_allocate load_addr = 0x15f000, Object # = 2
kmem_allocate load_addr = 0x161000, Object # = 3
Find_modules exe name = /pl2.gts
Find_modules exe name = /tsm_appl
kmem_allocate load_addr = 0x165000, Object # = 1
kmem_allocate load_addr = 0x169000, Object # = 2
kmem_allocate load_addr = 0x16b000, Object # = 3
...
```

Figure B.3: Example output from Harvester.vp script.

The command pulls the symbols from segment one of the map file and applies the linear offset of 0x0015b000. This command must be repeated for every segment in each module. Since VProbes only allows one symbol file per VM, subsequent segments should be added to the same symbol file.

APPENDIX C:
VProbe Generator GUI

The VProbe Generator GUI is a collection of Java tools for assisting users in accomplishing the tasks mentioned in this thesis. There are four main components of the VProbe Generator GUI: Control Pane, VProbe.out tail, VPEditor, and Symbol Table Generator. Due to the length of the source code it will not be included in this thesis. It has been archived by the CISR group at the Naval Postgraduate School. Each of the parts of this tool kit will be presented and the functionalities will be briefly discussed.

C.1 Control Pane

The control pane provides a general purpose interface for instrumenting the entire toolkit. The Symbol Generator, VP Editor, and VProbe.out Tail can all be started from this interface. Furthermore, this interface provides the ability to control VMware Virtual Machines. This includes; starting Virtual Machines, stopping Virtual Machines, and VProbe control.

The current implementation of the Control Panel has the ability to issue commands via the `vmrun` command. This provides the ability to stop and start virtual machines as well as perform VProbe-specific commands. Currently the following VProbe commands are supported: list globals, version, load probe, probe points and disable probes. These commands are available in the VProbe menu from the Control Panel, see Figure C.1. To use the control pane a target `.vmx` file must be specified. This provides a target for all of the `vmrun` commands.

A screen shot of the tool can be seen in Figure C.1.

C.2 Symbol Generator

The Symbol Generator tool allows users to build `nm` style symbol tables. This tool can parse a given MAP file and apply the offset. Saving the results to a file will append the contents to the selected file. If no file exists, one will be created.

A screen shot of the tool can be seen in Figure C.2. The Symbol Generator is available from the VProbe menu in the Control Panel.

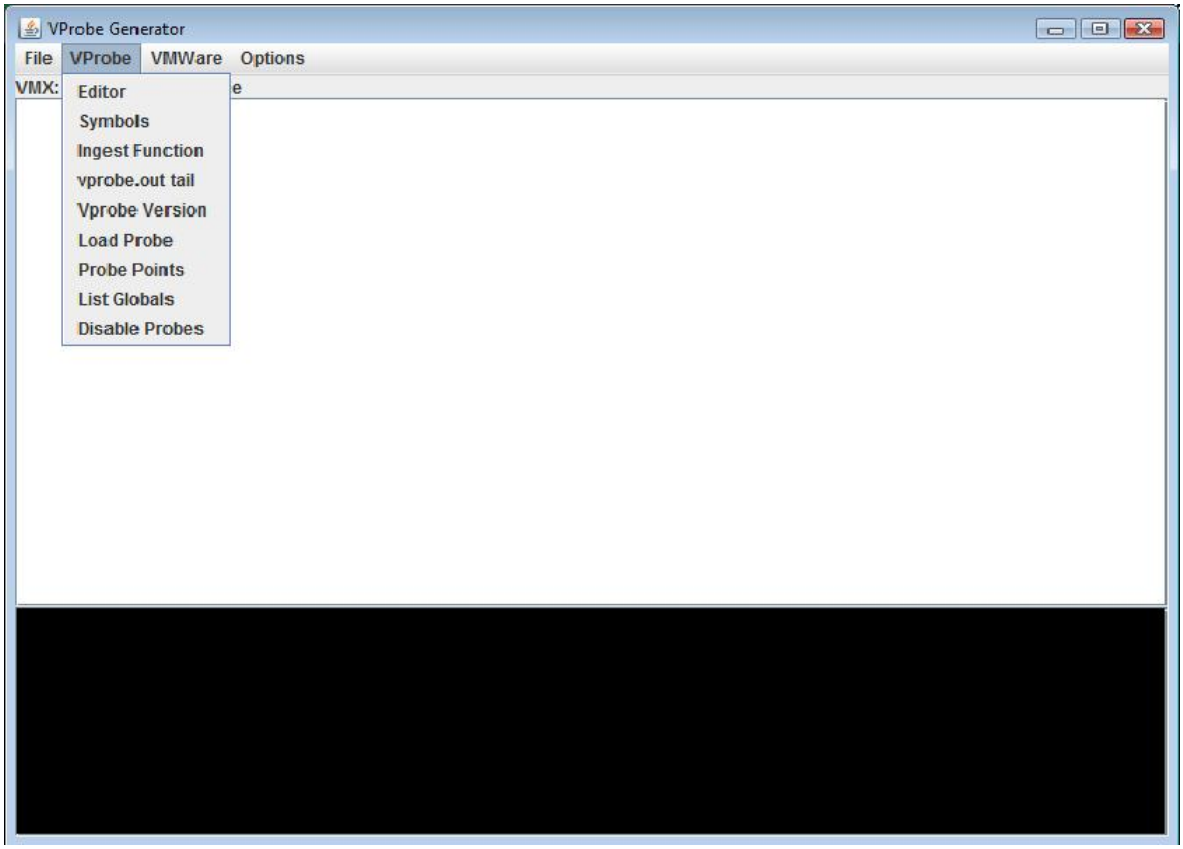

Figure C.1: Screen shot of the Control Pane from the VProbe Generator GUI

C.2.1 Using The Symbol Table Generator

It is assumed that a user reading this section has access to all the MAP files created during the compilation process. Furthermore, the `harvester.vp` script has been run and the resultant vprobe.out file is available. Finally, the user knows how to extract the appropriate information from these files. The subjects mentioned in this paragraph have been covered in detail elsewhere in this thesis.

The `Map File` field should specify the name of the MAP file to build the symbol table for. This can be entered directly into the text field or by browsing. The `Browse` will cause a `FileChooser` to open to allow the user to browse for the file. The path of the file should be relative to the current working directory in which the VProbe Generator GUI was started. It is recommended that the absolute path always be used as the current working directory can be specific to each system.

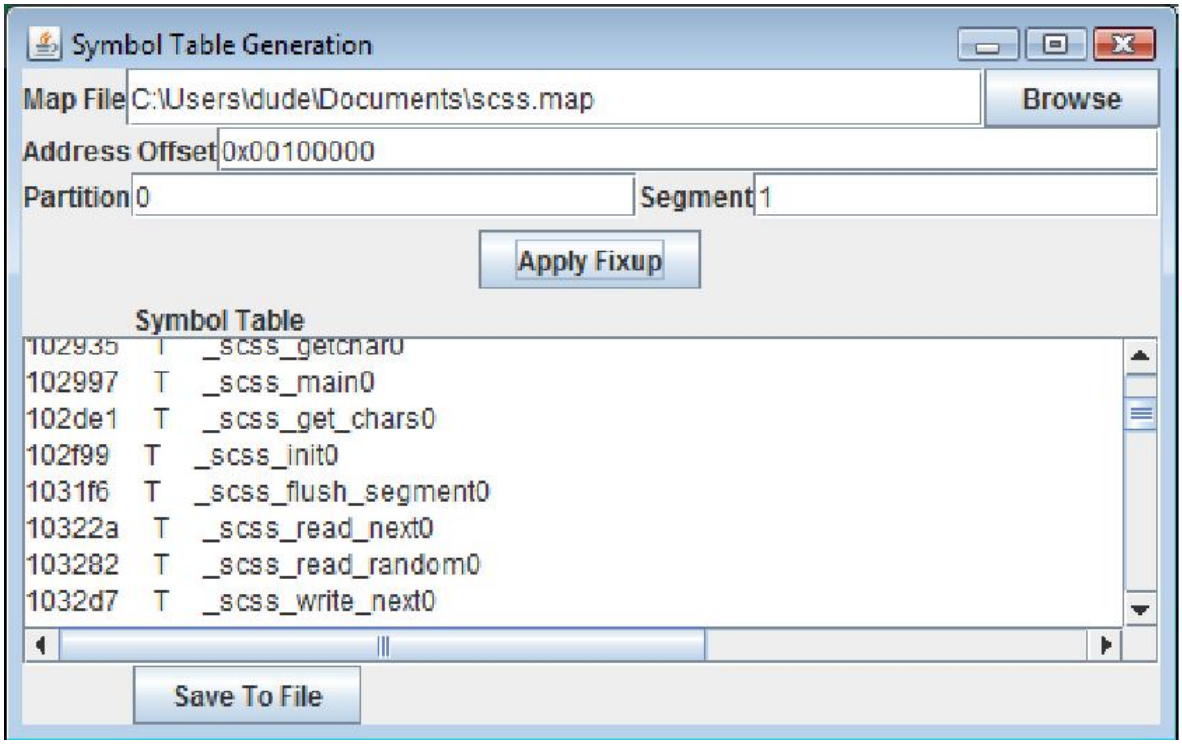

Figure C.2: Screen shot of the Symbol Generator

The `Address Offset` field specifies the linear address offset to be applied to the symbols. This information can be harvested from the results of the `vprobe.out` file. The format of this field should be a hex ascii string preceded by `0x`. For example, Figure C.2 shows a value of `0x00100000` which implies a linear address offset of 100000 hex bytes. This tool will only work for 32 bit addresses as the TCX kernel on works in 32 bit mode.

The `Partition` field specifies which partition the symbol resides in. The value specified will be appended to the end of the symbol name. For example, in Figure C.2 the partition value is 0. The resultant symbols in the `Symbol Table` field have 0 appended to the end of them.

The `Segment` field specifies which segment in the map file to pull symbols from. For example, in Figure C.2 a segment value of 1 is used. This implies that all symbols in the `symbol Table` field are from segment one in the map file.

C.3 VP Editor

The VP Editor provides a simple editor for creating VP Scripts. This tool will populate all possible probe points in the left panel. This includes any symbols from an imported symbol table.

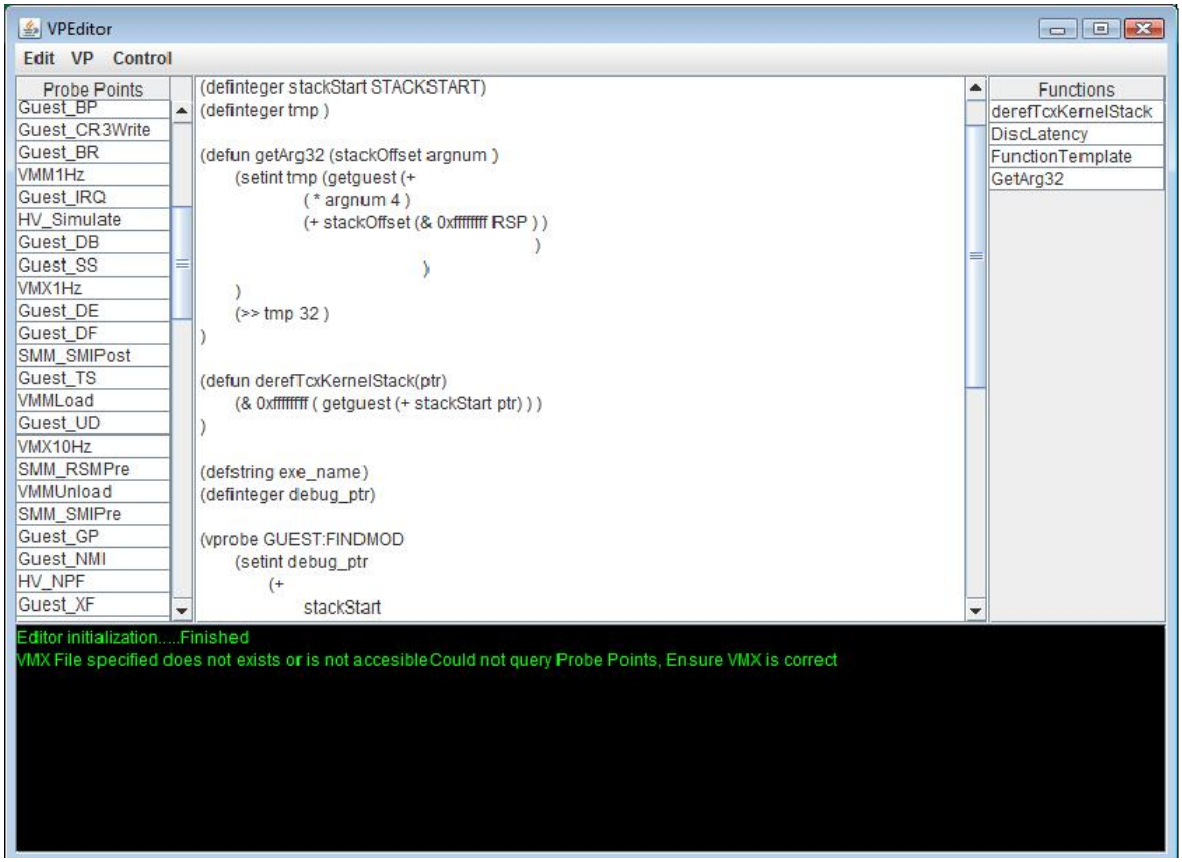

Figure C.3: Screen shot of the VP Editor

The panel to the right provides a simple mechanism for inserting user defined VP functions into the workspace.

A screen shot of the tool can be seen in Figure C.3. The VP Editor is accessible from the VProbe menu in the Control Panel.

C.4 VProbe.out Tail

The VProbe.out tail is modeled after the unix tool `tail`. It will watch the `vprobe.out` file for changes and update the workspace when any changes have been written. This tool also provides the ability to install hooks. Hooks are java classes that receive every line of input into the `vprobe.out` file.

The interface for writing hooks is defined in the `AutoHook.java` source file. Developers are required to install all hooks in the `/controller/hooks/` directory.

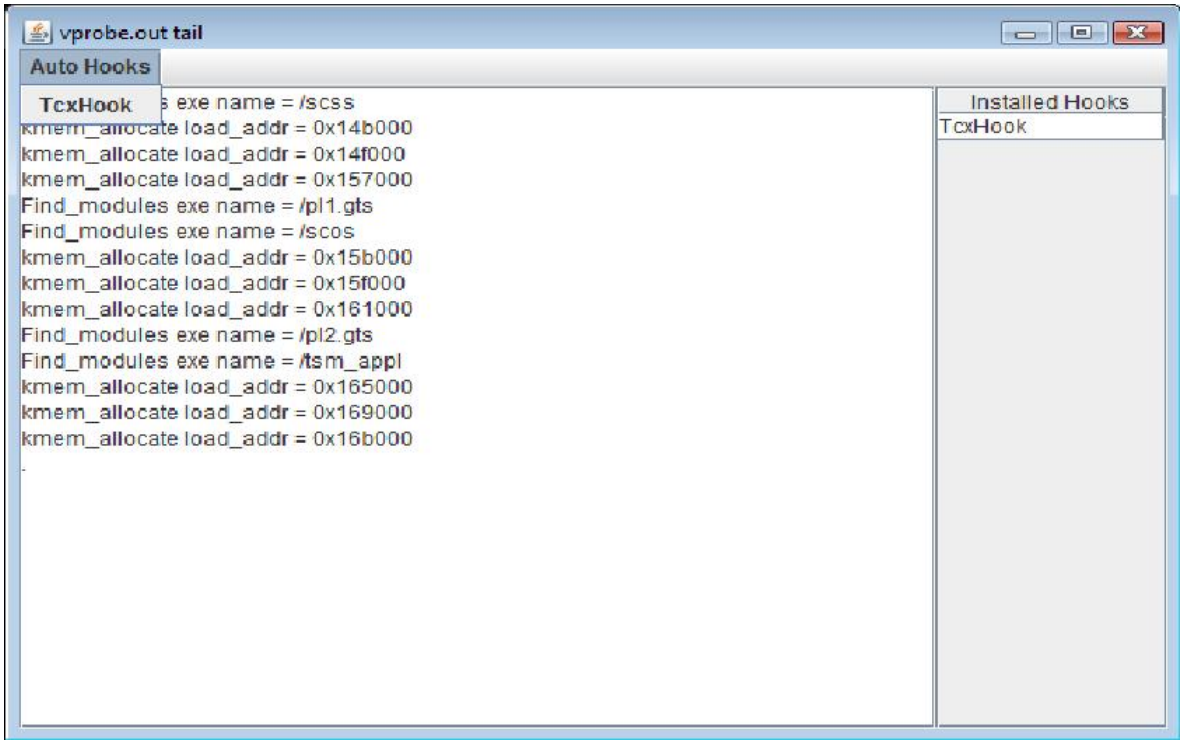

Figure C.4: Screen shot of the VProbe.out Tail Viewer

A screen shot of the tool can be seen in Figure C.4. The VProbe.out tail is available from the VProbe menu in the Control Panel.

C.4.1 Writing Hooks

Figure C.5 shows the source code for the TcxHook which can be seen in Figure C.4. The AutoHook interface defines 3 methods: init, hook, and setDebug. The init method is called immediately after object creation. The hook method is called every time the VProbe.out tail reads a line from the vprobe.out file. The line read from the vprobe.out file will be the parameter to the hook method. The setDebug method turns debugging on for the individual hook. This allows developers to dynamically turn on debugging code without compiling.

The TcxHook in Figure C.5 simply prints each line of input from the vprobe.out file to standard out. More extensive hooks can be created to perform tasks based on input received from VProbes.

```
package controller.hooks;
import controller.AutoHook;
public class TcxHook implements AutoHook
{

        private boolean debug = false;
        public boolean init()
        {
                this.debug = false;
                return true;
        }
        public boolean hook( String line )
        {
                System.out.println( line );
                return true;
        }

        public void setDebug( boolean val )
        {
                this.debug = val;
        }
}
```

Figure C.5: Example Hook code

APPENDIX D:
VPScript Listings

This chapter contains VP script listings used throughout the thesis. Each section in the Appendix represents a different VP script.

D.1 Vertical Test

```
;Code used for vertical test

; this function gets an argument from
;the stack based on the stack base address
( definteger tmp )
(defun getArg32 ( stackOffset argnum )
        ( setint tmp ( getguest
        (+( * argnum 4 ) (+ stackOffset (& 0 xffffffff RSP ) ) )
         ) )
        (>> tmp 32 )
)
; stack base address for
;scos (PL2), scss (PL1), and
;the kernel (PL0)
( definteger scos_stack 0x00161000)
( definteger scss_stack 0x00157000)
( definteger lpsk_stack 0x00120000)
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;; parameter values ;;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;
;get test_scos_gate param
(vprobe GUEST:_test_scos_gate0
        ( printf " test_scos_gate ( 0x%x ) \n"
                (getArg32 scos_stack 1)
        )
)
```

```
(vprobe GUEST:_scos_test0
        ( printf  " scos_test (  0x%x )\n"
                  (getArg32  scos_stack  1)
        )
)
;get  scss_gate
(vprobe GUEST:_test_scss_gate0
        ( printf  " test_scss_gate  (  0x%x ) \n"
                  (getArg32  scss_stack  1 )
        )
)
(vprobe GUEST:_scss_test0
        ( printf  " scss_test (  0x%x )\n"
                  (getArg32  scss_stack  1)
        )
)
;kernel  gate  call
(vprobe  GUEST:_test_lpsk_gate0
        ( printf  " test_lpsk_gate  (  0x%x )\n"
                  (getArg32  lpsk_stack  1 )
        )
)
(vprobe GUEST:_lpsk_test0
        ( printf  " lpsk_test (  0x%x )\n"
                  (getArg32  lpsk_stack  1)
        )
)
     ;;;;;;;;;;;;;;;;;;;;;;;;
 ;;;;  return  values  ;;;;;
     ;;;;;;;;;;;;;;;;;;;;;;;;

;vprobe  set  on  the  return  statement  of
;  test_scos_gate   function
(vprobe GUEST:0x0015b891
```

```
        ( printf " test_scos_gates = 0x%x\n" RAX)
)
;vprobe set on the return statement of
; scos_test function
(vprobe GUEST:0x0015b52d
        ( printf " scos_test = 0x%x\n" RAX )
)
;vprobe set on the return statement of
; test_scss_gate function
(vprobe GUEST:0x0014e469
        ( printf " test_scss_gate = 0x%x\n" RAX)
)
;vprobe set on the return statement of
; scss_test function
(vprobe GUEST:0x0014e219
        ( printf " scss_test = 0x%x\n" RAX )
)
;vprobe set on the return statement of
; test_lpsk_gate function
(vprobe GUEST:0x0010bfe4
        ( printf " test_lpsk_gates = 0x%x\n" RAX)
)
;vprobe set on the return statement of
; lpsk_test function
(vprobe GUEST:0x0010AAED
        ( printf " lpsk_test = 0x%x\n" RAX)
)
```

D.2 Horizontal Test

```
;code used for Horizontal test

; this function retrieves the value pointed to
by the parameter
(defun deref (ptr)
```

```
        (& 0 xffffffff ( getguest ptr ) )
)

;VProbe that watches the first_segment_variable in scss
; prints when the variable is read

(vprobe GUEST_READ:_first_segment_variable_scss0
        ( printf "ReadL first_segment_variable_scss ")
)

;vprobe that watches the first_segment variable in the lpsk
; prints when the cariable is read
(vprobe GUEST_READ:_first_segment_variable_lpsk0
        ( printf "ReadL first_segment_variable_lpsk ")
)

;vprobe that watches the second_segment_variable in scss
; prints the value of the variable when it is written to
(vprobe GUEST_WRITE:_second_segment_variable_scss0
        ( printf " _second_segment_variable_scss = 0x%x\n"
                (deref 0x163608)
        )
)
;vprobe that watches the second segment variable in the lpsk
; prints the value of the variable when it is written to
(vprobe GUEST_WRITE:_second_segment_variable_lpsk0
        ( printf " _second_segment_variable_lpsk = 0x%x\n"
                (deref 0x00120000)
        )
)
```

D.3 Invalid Write Test

```
;Code used with Invalid Write Test
```

```
(defun deref32 (ptr)
        (& 0 xffffffff ( getguest ptr ) )
)
; this vprobe watches the ISR for
; interrupt 13 ( invallid write)
(vprobe GUEST:0x001003a3 ;addr of INT13 handler
        ( printf "Firing ISR 13\n")
        ( printf "Version 0x%x\n" (getguest 0x00147000) )
)
;0x00147000 is the address of the read-only
;memory segment.
; this vprobe watches the first byte in the read-only
;memory segment
(vprobe GUEST_WRITE:0x00147000
        ( printf "Wrote to Memory segment value: 0x%x\n" (deref32 0x00147000) )
)
```

D.4 Harvester.vp

```
; Harvester.vp script

; replace STACKADDR with stack base address for the kernel
( definteger stackStart STACKADDR)

; this function gets an argument from
;the stack based on the stack base address
( definteger tmp )
(defun getArg32 ( stackOffset argnum )
        ( setint tmp ( getguest (+
                        ( * argnum 4 )
                        (+ stackOffset (& 0 xffffffff RSP ) )
                                )
                )
        )
        (>> tmp 32 )
```

```
)
;This function gets the value pointed to by the parameter
(defun derefTcxKernelStack( ptr )
        (& 0 xffffffff  ( getguest (+ stackStart  ptr ) ) )
)

( defstring  exe_name)
( definteger  debug_ptr )
;This probe watches execution for the find_module function
;when its executed the path path name is printed
(vprobe GUEST:FINDMOD ;replace FINDMOD with the address of find_module
        ( setint  debug_ptr
                (+
                        stackStart
                        (getArg32 stackStart  4 )
                )
        )
        ( getgeneststr  exe_name debug_ptr  )
        ( printf  "Find modules exe name = %s \n" exe_name)
)
;This vprobe watches the return statement in kmem_allocate.
;when it is executed the objet number and address are printed
( definteger  object_idx  0)
(vprobe GUEST:KMEMALLOC ;replace with return statement address for kmem_allocate
        ( setint  object_idx
                ;RBP-52 is the address of the object_idx .
                ;This is the segment number of the current segment
                ;This was found using IDA
                (derefTcxKernelStack (- RBP 52) )
        )
        ( printf  "EBP 0x%x\n" RBP )
        ( setint  object_idx  (+ object_idx  1 ) )
        ( printf  "kmem_allocate load_addr = 0x%x, Object # = %d\n"
                ( derefTcxKernelStack (getArg32 stackStart  3 ) )
```

object_idx

)

)

THIS PAGE INTENTIONALLY LEFT BLANK

Referenced Authors

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Defense Technical Information Center
   Fort Belvoir, Virginia

4. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

5. Susan Alexander
   OASD/NII DOD/CIO
   Washington, DC

6. George Bieber
   OSD
   Washington, DC

7. Kris Britton
   National Security Agency
   Fort Meade, MD

8. Ed Bryant
   Unified Cross Domain Management Office
   Maryland

9. John Campbell
   National Security Agency
   Fort Meade, MD

10. Deborah Cooper
    DC Associates, LLC
    Roslyn, VA

11. Dr. Steven C. Cooper
    National Science Foundation
    Arlington, VA

12. Grace Crowder
    National Security Agency
    Fort Meade, MD

13. Louise Davidson
    National Geospatial Agency
    Bethesda, MD

14. Steve Davis
    National Reconnaissance Office
    Chantilly, VA

15. Vincent J. DiMaria
    National Security Agency
    Fort Meade, MD

16. Rob Dobry
    National Security Agency
    Fort Meade, MD

17. Jennifer Guild
    SPAWAR
    Charleston, SC

18. CDR Scott Heller
    SPAWAR
    Charleston, SC

19. Steve LaFountain
    National Security Agency
    Fort Meade, MD

20. Dr. Greg Larson
    Institute for Defense Analyses
    Alexandria, VA

21. Dr. Karl Levitt
    National Science Foundation
    Arlington, VA

22. Dr. John Monastra
    Aerospace Corporation
    Chantilly, VA

23. John Mildner
    SPAWAR
    Charleston, SC

24. Dr. Victor Piotrowski
    National Science Foundation
    Arlington, VA

25. Jim Roberts
    Central Intelligence Agency
    Reston, VA

26. Ed Schneider
    Institute for Defense Analyses
    Alexandria, VA

27. Mark Schneider
    National Security Agency
    Fort Meade, MD

28. Keith Schwalm
    Good Harbor Consulting, LLC
    Washington, DC

29. Ken Shotting
    National Security Agency
    Fort Meade, MD

30. CDR Wayne Slocum
    SPAWAR
    San Diego, CA

31. Dr. Ralph Wachter
    Office of Naval Research
    Arlington, VA

32. Dr. Cynthia E. Irvine
    Naval Postgraduate School
    Monterey, CA

33. David Shifflett
    Naval Postgraduate School
    Monterey, CA

34. Boyd Fletcher
    SPAWAR
    San Diego, CA

35. Kyle M. Sanders
    Affiliation (SFS students: Civilian, Naval Postgraduate School)
    Monterey, CA